Escuela Superior de Ingeniería

Programación en Internet

Grado en Ingeniería Informática

Creación de un servicio Web REST y su despliegue en Tomcat

Autores:
Javier Montes Cumbrera y Salvador Carmona Román

Supervisores:
Juan Boubeta Puig y Guadalupe Ortiz Bellot

Cádiz, 29 de abril de 2015

Índice general

Índice de figuras

1. Creación del servicio

En este tutorial se describe la creación de un servicio Web REST y su despliegue en el servidor Tomcat.

1.1. Software necesario

Para poder crear nuestro servicio web *RESTful* tendremos que tener instalado una serie de software. El sistema operativo para poder seguir este tutorial es indiferente por lo que podemos utilizar el que tengamos, mientras sea compatible con las herramientas que vamos a utilizar.

El software necesario para seguir este tutorial será el siguiente:

- Java JDK [3].
- Eclipse JEE Luna [2].
- Apache Tomcat 8 [1].
- Bibliotecas Jersey + Jackson.
- Simple REST Client para Google Chrome [4].

1.2. Instalación y configuración

En este apartado vamos a instalar y configurar todo el software que nos hará falta y que hemos indicado en el apartado anterior.

1.2.1. Instalación de Java JDK

En este apartado vamos a realizar una diferenciación entre el sistema operativo de Microsoft y una distribución de Ubuntu.

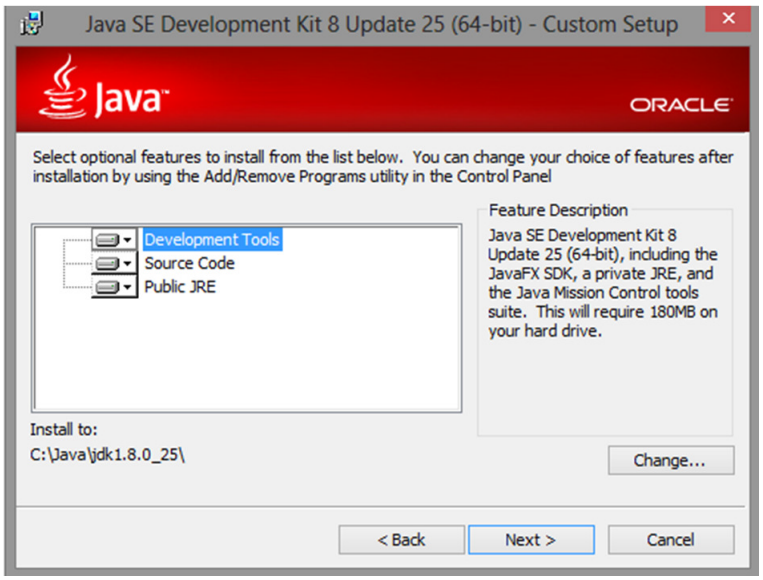

Figura 1.1.: Paso Inicial del Instalador Java JDK

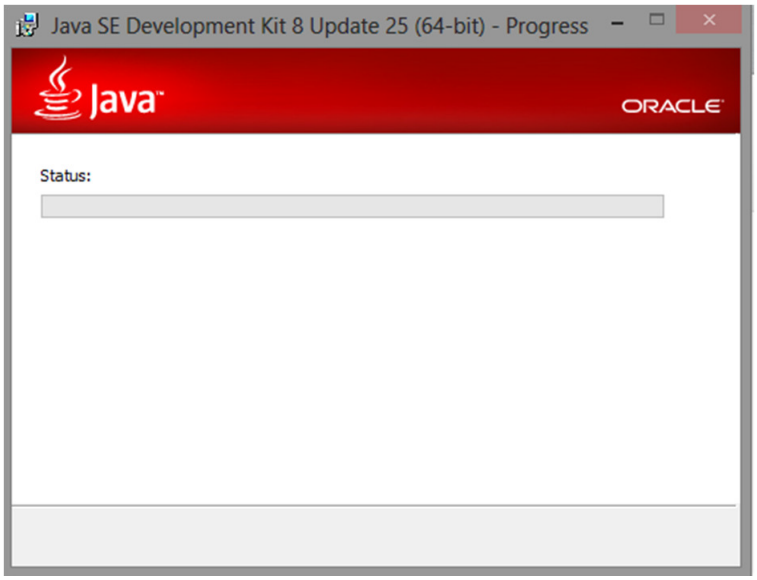

Figura 1.2.: Barra de instalación de Windows

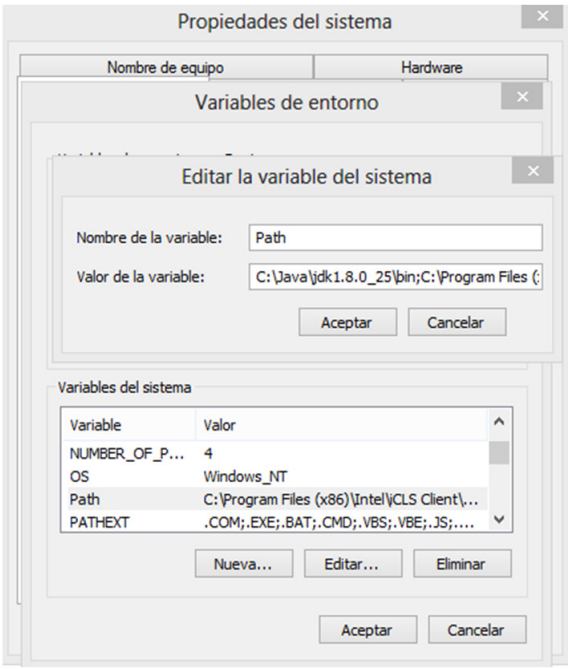

Figura 1.3.: Muestra de variables del entorno

Instalación en Windows

Esta versión es la más fácil de instalar, ya que la mayor parte de la instalación se hará automáticamente si seguimos los pasos del instalador, como podemos ver en la Figura 1.1 y la Figura 1.2.

Una vez instalado, tenemos que agregar la ruta del directorio 'bin' del JDK que acabamos de instalar en el Path del sistema. Para ello tendremos que hacer clic derecho sobre Equipo y elegimos la opción Propiedades en el menú desplegable que veremos. Después de que se abra el panel de propiedades buscaremos la pestaña de Opciones avanzadas y tendremos que pulsar sobre «variables de entorno», buscamos la variable Path entre todas las que tendrá el sistema. Una vez que la tengamos localizada la seleccionamos y pulsamos sobre el botón «editar», esto hará que se nos abra una ventana parecida a la de la Figura 1.3 en la que ya podremos, por fin, introducir nuestra ruta en el apartado «valor de la variable».

Es importante no borrar el contenido de esta ruta ya que podemos alterar el funcionamiento del sistema. Como podemos ver en la Figura 1.3 hemos optado por añadir nuestra ruta, C:\Java\jdk1.8.0_25\bin, añadiendo *;* al final.

Instalación en Linux

Una vez que hayamos descargado la versión para Linux, tendremos que descomprimir el contenido del fichero en la ruta /usr/local/java. Para hacer esto tendremos que movernos a la carpeta donde tengamos el fichero descargado y descomprimir el fichero con las órdenes del Listado 1.1.

Listado 1.1: Codigo para descomprimir el archivo Java

```
1 cd ~/Descargas
2 tar -xf <archivo> -C /usr/local/java
```

El siguiente paso sería decirle al sistema dónde se encuentra instalado el JDK de Java y cuál debe ser la ruta por defecto para llegar a él. En el Listado 1.2 podemos encontrar la secuencia de comandos necesarios para hacerlo.

Listado 1.2: Comandos necesarios para configurar Java

```
1 sudo -i
2 update-alternatives --install "/usr/bin/java" "java" "/usr/local/java/
    jdk1.8.0_25/bin/java" 1
3 update-alternatives --install "/usr/bin/javac" "javac" "/usr/local/java
    /jdk1.8.0_25/bin/javac" 1
4 update-alternatives --install "/usr/bin/javaws" "javaws" "/usr/local/
    java/jdk1.8.0_25/bin/javaws" 1
5 update-alternatives --set java /usr/local/java/jdk1.8.0_25/bin/java
6 update-alternatives --set java /usr/local/java/jdk1.8.0_25/bin/javac
7 update-alternatives --set java /usr/local/java/jdk1.8.0_25/bin/javaws
```

El último paso sería editar el fichero /etc/profile para añadir el Path de JAVA_HOME al final del ya existente, este paso se podría hacer con el editor de texto que prefiramos (véase el Listado 1.3).

Listado 1.3: Comando para editar el archivo profile

```
1 <editor de texto> /etc/profile
```

Cuando tecleemos la instrucción anterior tendrá que abrirnos un fichero, bajaremos hasta el final del documento e introduciremos las siguientes líneas del Listado 1.4 para que el sistema sepa cuál es la ruta de JAVA.

Listado 1.4: Modificaciones para el fichero /etc/profile

```
1 JAVA_HOME=/url/local/java/jdk1.8.0_25
2 PATH=$PATH:$HOME/bin:/$JAVA_HOME/bin
3 export JAVA_HOME
4 export PATH
```

1.2.2. Instalación de Eclipse JEE Luna

Tras descargar el software de la página oficial [2] solo tendremos que descomprimirlo en la ruta deseada, aunque se aconseja usar una ruta cercana a la raíz y sin espacios para evitar posibles problemas posteriores en un futuro. Un ejemplo podría ser C:\Development\eclipse en Windows y /home/development/eclipse en Linux.

1.2.3. Instalación de Apache Tomcat

Una vez tengamos descargado el fichero comprimido de Apache Tomcat de la página oficial [1] lo vamos a descomprimir siguiendo las mismas indicaciones que en la Sección 1.2.2. Para descomprimir el fichero comprimido bastará con teclear las líneas del Listado 1.5.

Listado 1.5: Comandos para descomprimir Tomcat

```
1 cd ~/Descargas
2 tar -xf <archivo> ~/development/tomcat
```

1.2.4. Configuración del servidor Tomcat

Una vez tenemos instalado todo lo necesario para trabajar con Eclipse y Tomcat es el momento de configurar el servidor Tomcat que soportará nuestro servicio y nos dará la visibilidad necesaria. Para lograr todo esto solo tendremos que seguir los siguientes pasos:

1. Una vez iniciado el Eclipse tendremos que seleccionar *File >> New >> Other* (véase Figura 1.4)

2. Seleccionar *Server* en la lista de elementos disponibles y hacemos clic en *Next* (véase Figura 1.5)

3. Buscamos en la lista de servidores disponibles el Apache Tomcat, seleccionamos *Tomcat v8.0 Server* y hacemos clic en *Next* (véase Figura 1.6)

4. Llegados a este punto tenemos dos opciones, una es especificar la carpeta donde tenemos descomprimido los ficheros de nuestro servidor Tomcat o hacer clic en *Browser* e indicar dónde están los ficheros. Hemos optado por escribir nuestra ruta como se especificó en el apartado 1.2.3. En la pestaña de JDK tendrá que aparecer seleccionada una opción por defecto por lo que tendremos que seleccionar el que hemos instalado en la Sección 1.2.1. Podrá darse dos situaciones diferentes:

   a) Si el JDK está instalado:

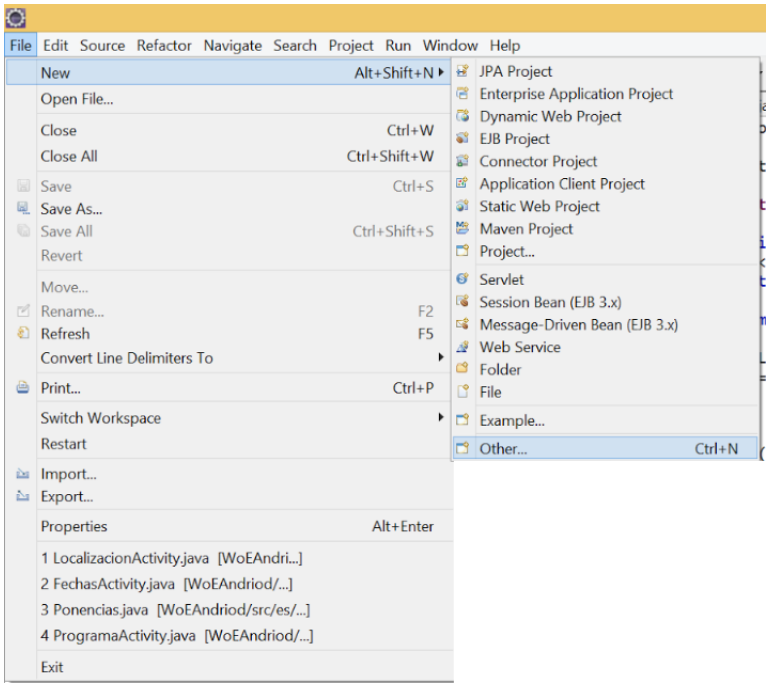

Figura 1.4.: Cuadro resultante de *File>>New>>Other*

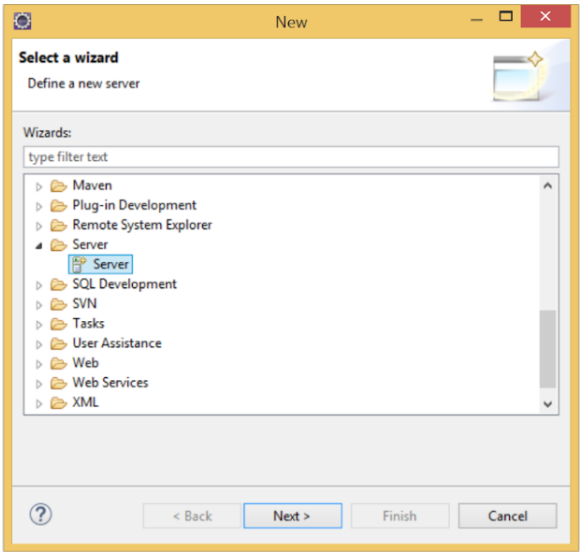

Figura 1.5.: Selección de servidor Tomcat

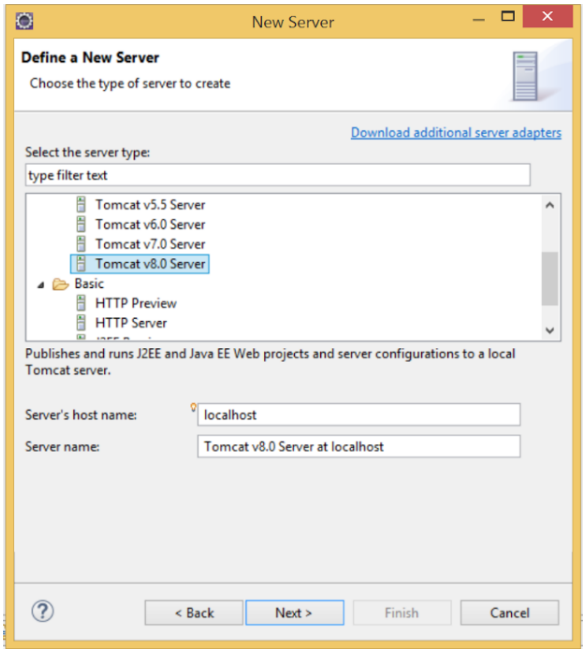

Figura 1.6.: Selección de servidor Tomcat

1) Hacemos clic en *Installed JREs...*

2) Marcamos el JDK que instalamos como se muestra en la Figura 1.7.

3) Hacemos clic en *Ok.*

4) Nos aseguramos que el JRE que aparece seleccionado es el JDK que acabamos de instalar.

b) Si el JDK no está instalado:

1) Hacemos clic en la siguiente secuencia *Installed JREs...>>Add.*

2) Seleccionamos la opción *Standar VM* y hacemos clic en *Next.*

3) En *JRE home* escribimos o buscamos la ruta donde instalamos el JDK en el paso 1.2.1, veremos algo parecido a la Figura 1.8, y hacemos clic en *Finish.*

4) Nos aseguramos que nuestro JDK queda seleccionado y volvemos a hacer clic en *Finish.*

5. Ya tendríamos que tener todo listo y tendríamos que poder ver algo parecido a la Figura 1.9. Ya solo nos queda hacer clic en *Finish* y tendríamos listo nuestro servidor Tomcat.

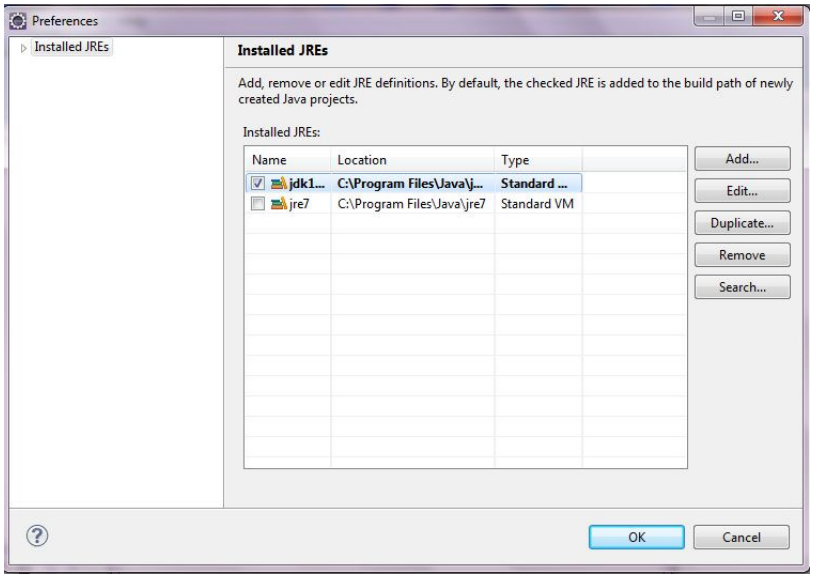

Figura 1.7.: Ejemplo de JDK seleccionado

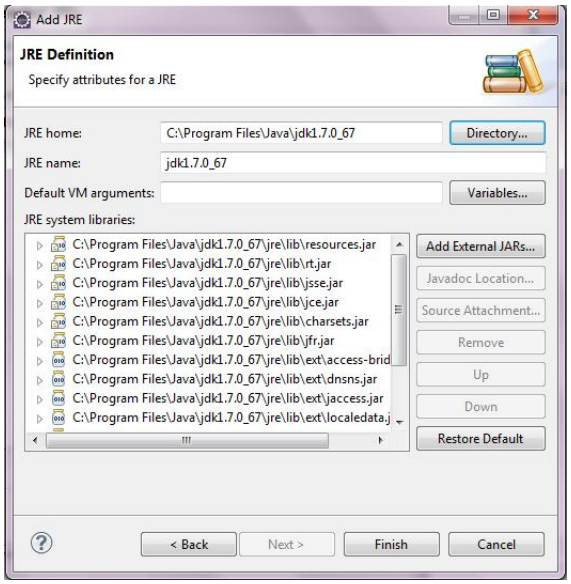

Figura 1.8.: Ejemplo de instalación de un nuevo JDK

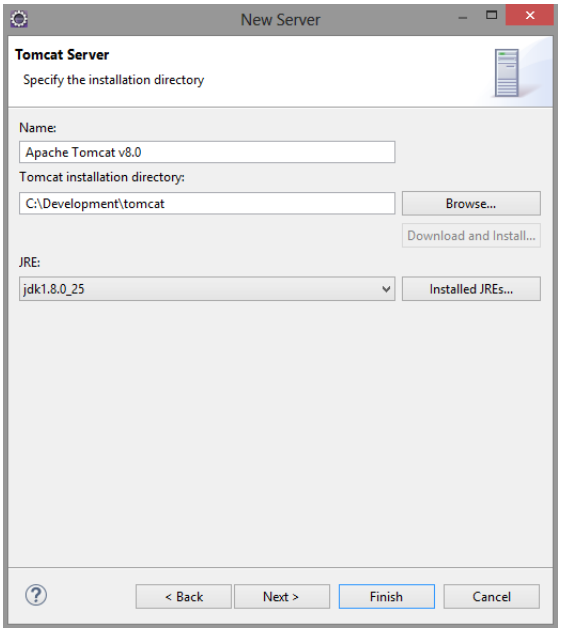

Figura 1.9.: Ejemplo de Tomcat configurado

Con estos pasos ya tendríamos nuestro servidor Tomcat preparado para poder ejecutar nuestro servicio web. Para lanzarlo solo tendríamos que hacer clic derecho sobre el proyecto, seleccionar la opción *Run As* y finalmente hacer clic en *Run on Server*.

1.3. Creación de un proyecto web dinámico

Para construir un servicio web tendremos que crear un proyecto web dinámico nuevo, para hacer esto solo tenemos que seleccionar *File>>New>>Dinamic Web Project* dentro de Eclipse (véase Figura 1.10)

Como nombre para el servicio web pondremos WoERest pero a la hora de la verdad dependerá del tipo de servicio que queramos ofrecer. Tenemos que prestar especial atención a las mayúsculas y a las minúsculas ya que es muy importante.

Una vez rellenada esta información podemos terminar haciendo clic en *Finish.*

1.4. Copiado de las bibliotecas de Jersey

Es fundamental en nuestro proyecto el uso de las bibliotecas que nos descargamos al principio del tutorial, primero tendremos que descomprimir el fichero que contiene las bibliotecas necesarias tal y como hicimos en la Sección 1.2.2. Una vez

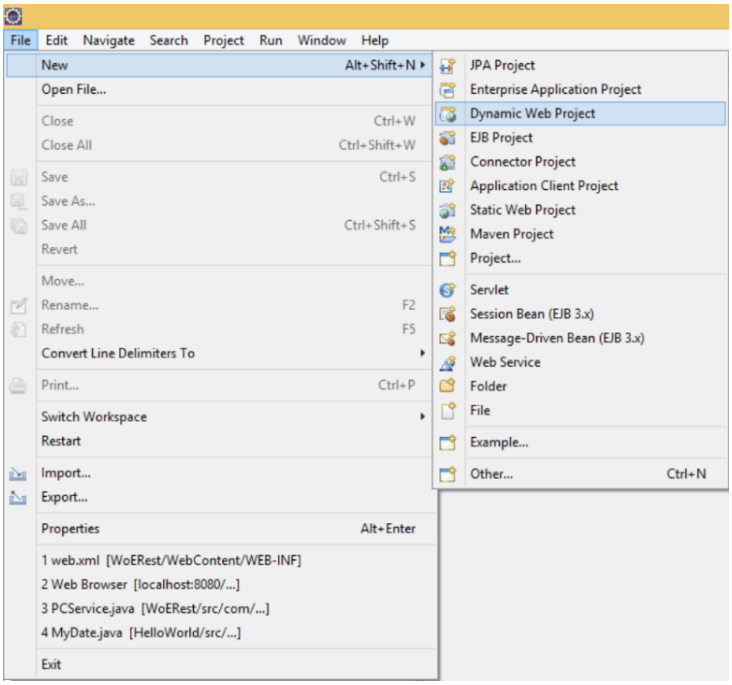

Figura 1.10.: Creación de un proyecto de web dinámico

descomprimidas las copiaremos en la carpeta WebContent/WEB-INF/lib que podemos encontrar en nuestro proyecto quedando algo similar a lo que podemos ver en la Figura 1.11.

1.5. Creación del paquete y de la clase

Después de añadir las bibliotecas necesarias tenemos que empezar a crear las funcionalidades de nuestro servicio web. Para ello tendremos que crear un paquete que encapsule todas las funcionalidades de nuestro servicio y también tendremos que crear las clases necesarias para ello. Los pasos a seguir para la creación del paquete y de las clases serían los siguientes:

1. Hacer clic derecho en *Java Resources>>src* que se encuentra en la carpeta de nuestro proyecto.

2. En el menú desplegable que se muestra tendremos que hacer clic en *New>>Package* tal y como podemos ver en la Figura 1.12.

3. Al hacer clic se nos habrá abierto una nueva ventana en la que tendremos que poner un nombre que identifique a nuestro paquete, se recomienda no utilizar mayúsculas o al menos que la primera letra sea minúscula para prevenir

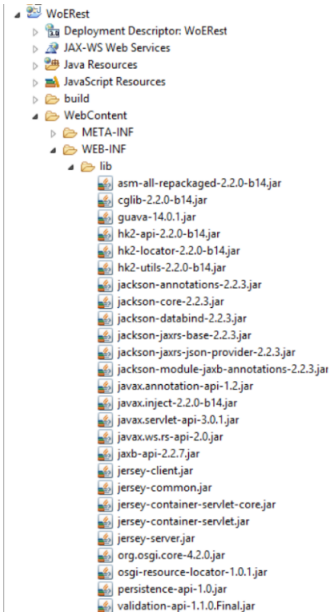

Figura 1.11.: Bibliotecas dentro del proyecto

errores y respetar las convenciones de Java. En nuestro caso vamos a utilizar com.woe.rest. Una vez establecido el nombre del paquete tendremos algo parecido a la Figura 1.13 y basta con que hagamos clic en Finish para terminar de crear nuestro paquete.

4. Una vez tenemos creado nuestro paquete tenemos que añadir una nueva clase para dotar de funcionalidad el servicio. Para ello tendremos que hacer clic derecho sobre el paquete que hemos creado antes y en el menú desplegable seleccionaremos *New>>Class*, como en el caso anterior tenemos que respetar las convenciones de Java por lo que tendremos que respetar las mayúsculas con las que empiezan todas las clases de Java. Esta clase será la que llamaremos PCService.java.

Para la creación de las funcionalidades del servicio necesitaremos dos clases adicionales, denominadas PCMember y PCDAO (véase el código en los Anexos A y B). Para poder seguir correctamente a partir de este punto tenemos que crear esas clases y copiar el código que se facilita. Aunque no vayamos a explicar cómo se crean esas clases sí vamos a explicar qué hacen cada una:

- PCMember: Esta clase permite modelar un miembro de un comité del programa de una conferencia con las 3 propiedades que hemos decidido que tenga, estas propiedades serían el nombre completo, institución a la que pertenece

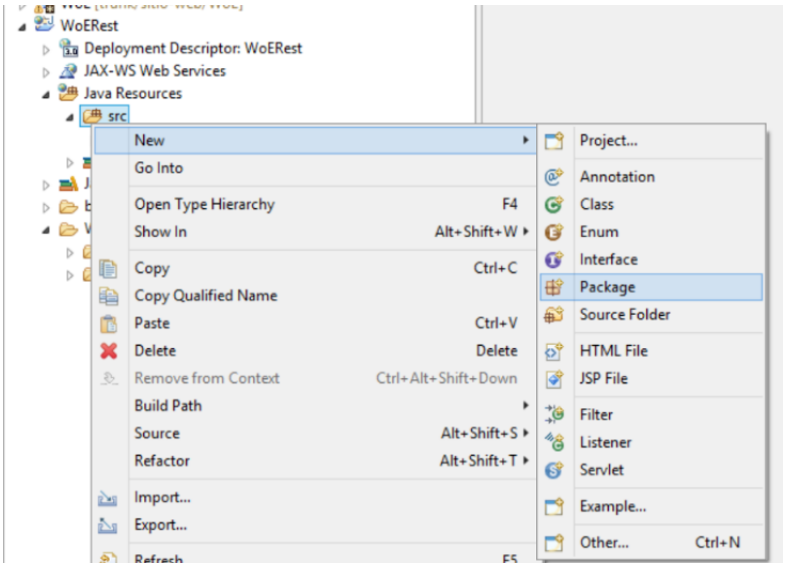

Figura 1.12.: Creando un nuevo paquete

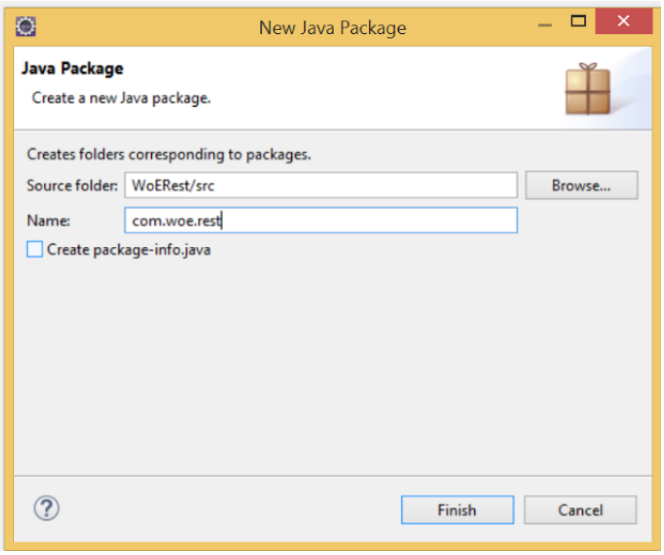

Figura 1.13.: Dando nombre al nuevo paquete

y su nacionalidad. En la clase también hemos añadido los métodos *setters* y *getters* correspondientes a los atributos para poder manejarlos.

- PCDAO: En esta clase nos encontramos las funcionalidades que se encargarán de gestionar los datos almacenados, alta, modificación, consulta y eliminación de miembros.

1.6. Añadiendo métodos al servicio

Como hemos dicho anteriormente la clase que albergará las funcionalidades es nuestra clase PCService, por lo que todas las funcionalidades que vamos a implementar ahora serán escritas dentro de esta.

1.6.1. Estableciendo la ruta hasta nuestro servicio

Antes de darle contenido a nuestra clase, tendremos que indicar con la etiqueta @Path la ruta para llegar a esta. Como podemos ver en el Listado 1.6, hemos elegido que la ruta para llegar hasta nuestra clase sea \Rest.

Listado 1.6: Clase PCService recién creada

```
1 package com.woe.rest;
2
3 @Path("/Rest")
4 public class PCService {
5 }
```

1.6.2. Instanciando la base de datos

La primera línea que nos encontremos en nuestra clase será la instanciación de la base de datos con la que trabajará el servicio. Como hemos dicho en la Sección 1.5, nuestra clase PCDAO será la encargada de controlar esto, por lo tanto tendremos que crear una instancia de esta clase (véase el Listado 1.7).

Listado 1.7: Primera línea de nuestro servicio

```
1 private PCDAO pcdao = new PCDAO();
```

1.6.3. GET

Después de haber creado una instancia del PCDAO podemos utilizarla, lo primero que haremos será implementar un método que devuelva la lista completa de los

miembros que componen nuestra base de datos en formato JSON a través de la URL del servicio añadiendo /allMembers al final (véase el Listado 1.8).

Listado 1.8: Código de nuestro método GET sin parámetros

```
1  @GET
2  @Path("/allMembers")
3  @Produces("application/json")
4  public List<PCMember> getAllPCMembers() throws Exception {
5      return pcdao.getAllMembers();
6  }
```

Como podemos ver el código es muy corto gracias a que el grueso de la lógica reside en el PCDAO. Nuestro método tiene 3 etiquetas adicionales que hacen que tenga la funcionalidad que nosotros queremos:

- @GET: esta etiqueta especifica que el método que estamos creando responderá a peticiones HTTP de este tipo y no a ninguna de otro tipo.
- @Path: la etiqueta Path establece la manera de llegar a este método a través de la URL de nuestro navegador.
- @Produces: esta etiqueta notificará al navegador que el método devolverá un tipo JSON como estructura —existen muchos otros tipos de los cuales veremos ejemplos más adelante—.

Dentro del tipo de peticiones GET también podemos permitir el paso de parámetros a través de la propia URL del navegador (véase el Listado 1.9).

Listado 1.9: Código de nuestro método GET con parámetros

```
1  @GET
2  @Path("/member/{memberKey}")
3  @Produces(MediaType.TEXT_PLAIN)
4  public String getMember(@PathParam("memberKey") String memberKey)
       throws Exception{
5      PCMember member = null;
6      member = pcdao.getMember(memberKey);
7      if (member != null)
8          return member.toString();
9      else
10         return "No encontrado.";
11 }
```

Como podemos comprobar hemos utilizado prácticamente las mismas etiquetas que en el caso anterior, pero con la excepción de que en la etiqueta @Path hemos añadido a la URL entre llaves una referencia a lo que será nuestro parámetro de entrada. Para poder pasar parámetros a la función también es necesario que en los

parámetros de entrada lo notifiquemos con la etiqueta @PathParam e indicando cuáles de los parámetros que recibimos a través de la URL queremos tomar, en el caso de que hubiese varios.

Otro cambio que hemos realizado es el de la línea 3, donde hemos decidido devolver texto plano en vez de devolver otra vez un JSON. Si quisiéramos devolver un JSON solo tendríamos que cambiar esta línea por una como en la del ejemplo anterior.

1.6.4. PUT

Después de haber tratado con el tipo de peticiones más conocido del protocolo HTTP, vamos a crear un método que actualiza algún registro de nuestra base de datos.

Un punto muy importante que tenemos que tener en cuenta es que para que un JSON se transforme a un objeto Java, y viceversa, la clase Java tiene que poder ser serializada. Esto lo conseguimos con la etiqueta @XmlRootElement, esta etiqueta tiene que ir colocada antes de la definición de la clase que queremos que sea serializada, como ejemplo podría tomarse nuestra clase PCMember (véase el Listado 1.10).

Listado 1.10: Muestra de utilización de la etiqueta @XmlRootElement

```
1  package com.woe.rest;
2  import javax.xml.bind.annotation.XmlRootElement;
3  @XmlRootElement
4  public class PCMember{
5      private String nombre;
6      private String afiliacion;
7      private String nacionalidad;
8      .
9      .
10     .
11 };
```

Si queremos hacer un método PUT tendremos que indicarlo con la etiqueta @PUT y además tenemos qué indicar que tipo de dato esperamos recibir desde el navegador. Para esto utilizaremos la etiqueta @Consumes junto al tipo de dato que queremos recibir (véase el Listado 1.11).

Listado 1.11: Código de nuestro método PUT

```
1  @PUT
2  @Path("/update/{memberKey}")
3  @Consumes(MediaType.APPLICATION_JSON)
4  @Produces(MediaType.TEXT_PLAIN)
5  public String updateMember(@PathParam("memberKey") String memberKey,
       PCMember member) throws Exception{
6      if(pcdao.updateMember(memberKey, member))
7          return "Actualizado con exito.";
```

```
8     else
9         return "No se pudo actualizar.";
10 }
```

Como podemos ver en el código, no hemos notificado en los parámetros de entrada el objeto JSON que vamos a recibir, a diferencia de como lo hacemos con la etiqueta @Path. Esto se debe a que para recibir datos en general desde el navegador basta con añadir en los parámetros de entrada un objeto de la clase a la cual queremos convertir la información.

1.6.5. DELETE

Otro método que podemos incluir en nuestro servicio es el DELETE, como bien podemos imaginarnos este tipo de peticiones sirve para borrar. Para hacer un método de estas características solo tenemos que añadir la etiqueta @DELETE como en el anterior de los casos y ya tenemos nuestro método que borra un miembro determinado (véase el Listado 1.12).

Listado 1.12: Código de nuestro método DELETE

```
1  @DELETE
2  @Path("/delete/{memberKey}")
3  @Produces({ MediaType.TEXT_PLAIN })
4  public String deleteMember(@PathParam("memberKey") String memberKey)
       throws Exception {
5      if(pcdao.deleteMember(memberKey))
6          return "Eliminado.";
7      else
8          return "No se pudo eliminar.";
9  }
```

1.6.6. POST

Tras probar las etiquetas GET, PUT y DELETE, falta por describir la etiqueta POST. Esta etiqueta sirve para crear elementos nuevos dentro del servicio y se utiliza de forma similar a las otras etiquetas. Para utilizarlo tenemos que utilizar la etiqueta @POST e indicar cómo le pasaremos los parámetros, la primera forma que utilizaremos es enviando un JSON como hemos hecho anteriormente (véase el Listado 1.13).

Listado 1.13: Código de nuestro método POST

```
1  @POST
2  @Path("/new/{memberKey}")
3  @Consumes({ "application/json"})
4  @Produces(MediaType.TEXT_PLAIN)
```

```
5  public String newMember(@PathParam("memberKey") String memberKey,
       PCMember member) throws Exception {
6      if(pcdao.newMember(memberKey, member))
7          return "Miembro agregado.";
8      else
9          return "Miembro no agregado.";
10 }
```

La segunda forma de utilizarlo es enviando los datos desde un formulario creado para tal efecto, en este caso tendremos que indicar en la lista de argumentos de entrada cuáles son los parámetros que le pasamos mediante el formulario y el nombre que le pusimos en el mismo. Para esto se utiliza la etiqueta @FormParam acompañado del nombre del campo que queremos utilizar (véase el Listado 1.14).

Listado 1.14: Método POST con paso de parámetros mediante formulário web

```
1  @POST
2  @Path("/new")
3  @Consumes("application/x-www-form-urlencoded")
4  @Produces(MediaType.TEXT_PLAIN)
5  public String newMemberForm(@FormParam("memberKey") String memberKey,
       @FormParam("memberName") String nombre, @FormParam("memberAfil")
       String afiliacion, @FormParam("memberNation") String nacionalidad)
       throws Exception {
6      PCMember member = new PCMember();
7      member.setNombre(nombre);
8      member.setAfiliacion(afiliacion);
9      member.setNacionalidad(nacionalidad);
10     if(pcdao.newMember(memberKey, member))
11         return "Miembro agregado.";
12     else
13         return "Miembro no agregado.";
14 }
```

Como podemos ver en el Listado 1.14, la etiqueta @Consumes notifica que obtendremos datos a través de un formulario web y con la etiqueta @FormParam estamos diciendo qué valor corresponde entre el formulario recibido y el parámetro de entrada.

1.7. Despliegue del servicio en Tomcat

Llegados a este punto prácticamente tenemos un servicio completo, pero falta una cosa muy importante, el archivo de configuración XML denominado *web.xml*. Este fichero es necesario para el despliegue del servicio en Tomcat, dicho fichero tiene que encontrarse en la carpeta WEB-INF, donde introducimos las bibliotecas en el

paso 1.4. Para crear el fichero tendremos que hacer clic derecho en la carpeta WEB-INF y luego en *New>>XML File*. Una vez creado el fichero, tendremos que copiar las siguientes líneas sustituyendo las partes importantes:

NOMBRE_DEL_SERVICIO: Un nombre identificador del servicio que en este caso será WoERest.

NOMBRE_DE_TU_PAQUETE: Nombre del paquete donde está implementado el servicio web que en este caso es com.woe.rest.

Algo en lo que tenemos que prestar especial atención es en la línea 18 del Listado 1.15. En esta línea estamos indicando a través de qué URL permitiremos el recibir peticiones a nuestro servicio. Tal y como está en el código permitiría llamadas desde *localhost:8080/WoERest.*

Listado 1.15: Código de nuestro fichero web.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3  xmlns="http://java.sun.com/xml/ns/javaee"
4  xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
5  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
6  http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
7  id="WebApp_ID" version="3.0">
8  <servlet>
9      <servlet-name>WoERest</servlet-name>
10     <servlet-class>org.glassfish.jersey.servlet.ServletContainer</
           servlet-class>
11     <init-param>
12         <param-name>jersey.config.server.provider.packages</param-name>
13         <param-value>com.woe.rest,com.fasterxml.jackson.jaxrs.json</param
               -value>
14     </init-param>
15 </servlet>
16 <servlet-mapping>
17     <servlet-name>WoERest</servlet-name>
18     <url-pattern>/*</url-pattern>
19 </servlet-mapping>
20 </web-app>
```

1.8. Iniciando el Servicio

Después de instalar todo el software necesario, configurar el servidor Tomcat, añadir las bibliotecas necesarias e implementar todas las funciones, tendremos nuestro servicio funcionando y preparado para recibir consultas desde un cliente web o cualquier otro dispositivo.

Bibliografía

[1] Apache Tomcat - Apache Tomcat 8 Downloads (2015), https://tomcat.apache.org/download-80.cgi

[2] Eclipse IDE for Java EE Developers | Packages (2015), https://eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/lunar

[3] Java SE Development Kit 8 - Downloads (2015), http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html?ssSourceSiteId=otnes

[4] Google: Simple REST Client (2015), https://chrome.google.com/webstore/detail/simple-rest-client/fhjcajmcbmldlhcimfajhfbgofnpcjmb

A. PCMember

En este anexo se muestra el código de la clase PCMember.

```
1  package com.woe.rest;
2
3  import javax.xml.bind.annotation.XmlRootElement;
4
5  @XmlRootElement
6  public class PCMember{
7     private String key;
8     private String nombre;
9     private String afiliacion;
10    private String nacionalidad;
11
12
13    public PCMember(){}
14    public PCMember(String key, String nombre, String afiliacion, String
          nacionalidad){
15
16       this.key = key;
17       this.nombre = nombre;
18       this.afiliacion = afiliacion;
19       this.nacionalidad = nacionalidad;
20    }
21
22    public String getNombre() {
23       return nombre;
24    }
25
26    public void setNombre(String nombre) {
27       this.nombre = nombre;
28    }
29
30    public String getAfiliacion() {
31       return afiliacion;
32    }
33
34    public void setAfiliacion(String afiliacion) {
35       this.afiliacion = afiliacion;
36    }
37
38    public String getNacionalidad() {
```

```
39          return nacionalidad;
40      }
41
42      public void setNacionalidad(String nacionalidad) {
43          this.nacionalidad = nacionalidad;
44      }
45
46      public String getKey(){
47          return key;
48      }
49
50      public void setKey(String key){
51          this.key = key;
52      }
53
54      public String toString(){
55          return nombre + ", " + afiliacion + ", " + nacionalidad;
56      }
57  }
```

B. PCDAO

En este anexo se muestra el código de la clase PCDAO.

```
1  package com.woe.rest;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  public class PCDAO{
7
8      private static List<PCMember> bd = new ArrayList<PCMember>();
9      static {
10         PCMember miembro1 = new PCMember("MarcoAiello", "Marco Aiello","
               Universidad de Groningen","Paises Bajos");
11         PCMember miembro2 = new PCMember("VasiliosAndrikopoulos", "
               Vasilios Andrikopoulos","Universidad de Stuttgart","Alemania"
               );
12         PCMember miembro3 = new PCMember("AntonioBrogi", "Antonio Brogi",
               "Universidad de Pisa","Italia");
13
14         bd.add(miembro1);
15         bd.add(miembro2);
16         bd.add(miembro3);
17     }
18
19     public List<PCMember> getAllMembers() {
20         return bd;
21     }
22
23     public PCMember getMember(String memberKey){
24       int keyMember=0;
25       for(int i = 0; i < bd.size(); i++)
26          if(bd.get(i).getKey().equals(memberKey))
27             keyMember = i;
28        return bd.get(keyMember);
29     }
30
31     public boolean updateMember(String memberKey, PCMember member){
32       int keyMember=0;
33        for(int i = 0; i < bd.size(); i++)
34          if(bd.get(i).getKey().equals(memberKey))
35             keyMember = i;
```

```
36          if(bd.get(keyMember).getKey().equals(memberKey)){
37              bd.set(keyMember,member);
38              return true;
39           }else
40            return false;
41      }
42      public boolean deleteMember(String memberKey){
43        int keyMember=0;
44        for(int i = 0; i < bd.size(); i++)
45            if(bd.get(i).getKey().equals(memberKey))
46                keyMember = i;
47         if(bd.get(keyMember).getKey().equals(memberKey)){
48            bd.remove(keyMember);
49            return true;
50         }else
51           return false;
52      }
53      public boolean newMember(String memberKey, PCMember member){
54          bd.add(member);
55          return true;
56      }
57  }
```